CHAPTER 6

Prepping Your Data with Pandas

With the explosion of the Internet, social networks, mobile devices, and big data, the amount of data available is humongous. Managing and analyzing this data to derive meaningful inferences can drive decision making, improve productivity, and reduce costs. In the previous chapter, you learned about NumPy – the library that helps us work with arrays and perform computations, also serving as the backbone for the Pandas library that we discuss in this chapter. Pandas, the Python library for data wrangling, has the advantage of being a powerful tool with many capabilities to manipulate data.

The rising popularity of Python as the preferred programming language is closely related to its widespread applications in the field of data science. In a survey conducted in 2019 among Python developers, it was found that NumPy and Pandas are the most popular data science frameworks (Source: https://www.jetbrains.com/lp/python-developers-survey-2019/).

In this chapter, we learn about the building blocks of the Pandas (Series, DataFrames, and Indexes), and understand the various functions in this library that are used to tidy, cleanse, merge, and aggregate data in Pandas. This chapter is more involved than the other chapters you have read so far since we are covering a wide range of topics that will help you develop the skills necessary for preparing your data.

Pandas at a glance

Wes McKinney developed the Pandas library in 2008. The name (Pandas) comes from the term "Panel Data" used in econometrics for analyzing time-series data. Pandas has many features, listed in the following, that make it a popular tool for data wrangling and analysis.

G. Rajagopalan, *A Python Data Analyst's Toolkit*, https://doi.org/10.1007/978-1-4842-6399-0_6

1. Pandas provides features for labeling of data or indexing, which speeds up the retrieval of data.
2. Input and output support: Pandas provides options to read data from different file formats like JSON (JavaScript Object Notation), CSV (Comma-Separated Values), Excel, and HDF5 (Hierarchical Data Format Version 5). It can also be used to write data into databases, web services, and so on.
3. Most of the data that is needed for analysis is not contained in a single source, and we often need to combine datasets to consolidate the data that we need for analysis. Again, Pandas comes to the rescue with tailor-made functions to combine data.
4. Speed and enhanced performance: The Pandas library is based on Cython, which combines the convenience and ease of use of Python with the speed of the C language. Cython helps to optimize performance and reduce overheads.
5. Data visualization: To derive insights from the data and make it presentable to the audience, viewing data using visual means is crucial, and Pandas provides a lot of built-in visualization tools using Matplotlib as the base library.
6. Support for other libraries: Pandas integrates smoothly with other libraries like Numpy, Matplotlib, Scipy, and Scikit-learn. Thus we can perform other tasks like numerical computations, visualizations, statistical analysis, and machine learning in conjunction with data manipulation.
7. Grouping: Pandas provides support for the split-apply-combine methodology, whereby we can group our data into categories, apply separate functions on them, and combine the results.
8. Handling missing data, duplicates, and filler characters: Data often has missing values, duplicates, blank spaces, special characters (like $, &), and so on that may need to be removed or replaced. With the functions provided in Pandas, you can handle such anomalies with ease.

9. Mathematical operations: Many numerical operations and computations can be performed in Pandas, with NumPy being used at the back end for this purpose.

Technical requirements

The libraries and the external files needed for this chapter are detailed in the following.

Installing libraries

If you have not already installed Pandas, go to the Anaconda Prompt and enter the following command.

```
>>>pip install pandas
```

Once the Pandas library is installed, you need to import it before using its functions. In your Jupyter notebook, type the following to import this library.

CODE:

```
import pandas as pd
```

Here, *pd* is a shorthand name or alias that is a standard for Pandas.

For some of the examples, we also use functions from the NumPy library. Ensure that both the Pandas and NumPy libraries are installed and imported.

External files

You need to download a dataset, "`subset-covid-data.csv`", that contains data about the number of cases and deaths related to the COVID-19 pandemic for various countries on a particular date. Please use the following link for downloading the dataset: `https://github.com/DataRepo2019/Data-files/blob/master/subset-covid-data.csv`

Building blocks of Pandas

The Series and DataFrame objects are the underlying data structures in Pandas. In a nutshell, a Series is like a column (has only one dimension), and a DataFrame (has two dimensions) is like a table or a spreadsheet with rows and columns. Each value stored in a Series or a DataFrame has a label or an index attached to it, which speeds up retrieval and access to data. In this section, we learn how to create a Series and DataFrame, and the functions used for manipulating these objects.

Creating a Series object

The Series is a one-dimensional object, with a set of values and their associated indexes. Table 6-1 lists the different ways of creating a series.

Table 6-1. *Various Methods for Creating a Series Object*

METHOD	SYNTAX

Using a scalar value	CODE (for creating a Series using a scalar value):
	`pd.Series(2)`
	`#Creating a simple series with just one value. Here, 0 is the index label, and 2 is the value the Series object contains.`
	Output:
	`0 2`
	`dtype: int64`

Using a list	CODE (for creating a Series using a list):
	`pd.Series([2]*5)`
	`#Creating a series by enclosing a single value (2) in a list and replicating it 5 times. 0,1,2,3,4 are the autogenerated index labels.`
	Output:
	`0 2`
	`1 2`
	`2 2`
	`3 2`
	`4 2`
	`dtype: int64`

Using characters in a string	CODE (for creating a series using a string):
	`pd.Series(list('hello'))`
	`#Creating a series by using each character in the string "hello" as a separate value in the Series.`

(*continued*)

Table 6-1. (*continued*)

METHOD	SYNTAX

	Output:
	`0 h`
	`1 e`
	`2 l`
	`3 l`
	`4 o`
	`dtype: object`

Using a dictionary	CODE (for creating a Series from a dictionary):
	`pd.Series({1:'India',2:'Japan',3:'Singapore'})`
	#The key/value pairs correspond to the index labels and values in the Series object.
	
	Output:
	`1 India`
	`2 Japan`
	`3 Singapore`
	`dtype: object`

Using a range	CODE (for creating a Series from a range):
	`pd.Series(np.arange(1,5))`
	`#Using the NumPy arrange function to create a series from a range of 4 numbers (1-4), ensure that the NumPy library is also imported`
	
	Output:
	`0 1`
	`1 2`
	`2 3`
	`3 4`
	`dtype: int32`

(*continued*)

Table 6-1. (*continued*)

METHOD	SYNTAX

Using random numbers	CODE (for creating a Series from random numbers):
	`pd.Series(np.random.normal(size=4))`
	`#Creating a set of 4 random numbers using the np.random.normal function`
	Output:
	`0 -1.356631`
	`1 1.308935`
	`2 -1.247753`
	`3 -1.408781`
	`dtype: float64`

Creating a series with customized index labels	CODE (for creating a custom index):
	`pd.Series([2,0,1,6],index=['a','b','c','d'])`
	`#The list [2,0,1,6] specifies the values in the series, and the list for the index['a','b','c','d'] specifies the index labels`
	Output:
	`a 2`
	`b 0`
	`c 1`
	`d 6`
	`dtype: int64`

To summarize, you can create a Series object from a single (scalar) value, list, dictionary, a set of random numbers, or a range of numbers. The *pd.Series* function creates a Series object (note that the letter "S" in "Series" is in uppercase; pd.series will not work). Use the index parameter if you want to customize the index.

Examining the properties of a Series

In this section, we will look at the methods used for finding out more information about a Series object like the number of elements, its values, and unique elements.

Finding out the number of elements in a Series

There are three ways of finding the number of elements a Series contains: using the *size* parameter, the *len* function, or the *shape* parameter

The *size* attribute and the *len* function return a single value - the length of the series, as shown in the following.

CODE:

```
#series definition
x=pd.Series(np.arange(1,10))
#using the size attribute
x.size
```

Output:

```
9
```

We can also use the *len* function for calculating the number of elements, which would return the same output (9), as shown in the following.

CODE:

```
len(x)
```

The *shape* attribute returns a tuple with the number of rows and columns. Since the Series object is one-dimensional, the shape attribute returns only the number of rows, as shown in the following.

CODE:

```
x.shape
```

Output:

```
(9,)
```

Listing the values of the individual elements in a Series

The *values* attribute returns a NumPy array containing the values of each item in the Series.

CODE:

```
x.values
```

Output:

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Accessing the index of a Series

The index of the Series can be accessed through the index attribute. An index is an object with a data type and a set of values. The default type for an index object is *RangeIndex*.

CODE:

```
x.index
```

Output:

```
RangeIndex(start=0, stop=9, step=1)
```

The index labels form a range of numbers, starting from 0. The default step size or the difference between one index label value and the next is 1.

Obtaining the unique values in a Series and their count

The *value_counts()* method is an important method. When used with a Series object, it displays the unique values contained in this object and the count of each of these unique values. It is a common practice to use this method with a categorical variable, to get an idea of the distinct values it contains.

CODE:

```
z=pd.Series(['a','b','a','c','d','b'])
z.value_counts()
```

Output:

```
a    2
b    2
c    1
d    1
dtype: int64
```

The preceding output shows that in the Series object named "z", the values "a" and "b" occur twice, while the characters "c" and "d" occur once.

Method chaining for a Series

We can apply multiple methods to a series and apply them successively. This is called method chaining and can be applied for both Series and DataFrame objects.

Example:

Suppose we want to find out the number of times the values "a" and "b" occur for the series "z" defined in the following. We can combine the *value_counts* method and the *head* method by chaining them.

CODE:

```
z=pd.Series(['a','b','a','c','d','b'])
z.value_counts().head(2)
```

Output:

```
a    2
b    2
dtype: int64
```

If multiple methods need to be changed together and applied on a Series object, it is better to mention each method on a separate line, with each line ending with a backslash. It would make the code more readable, as shown in the following.

CODE:

```
z.value_counts()\
.head(2)\
.values
```

Output:

```
array([2, 2], dtype=int64)
```

We have covered the essential methods that are used with the Series object. If you want to learn more about the Series object and the methods used with Series objects, refer to the following link.

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html

We now move ovn to DataFrames, another important Pandas object.

DataFrames

A DataFrame is an extension of a Series. It is a two-dimensional data structure for storing data. While the Series object contains two components - a set of values, and index labels attached to these values - the DataFrame object contains three components - the column object, index object, and a NumPy array object that contains the values.

The index and columns are collectively called the axes. The index forms the axis "0" and the columns form the axis "1".

We look at various methods for creating DataFrames in Table 6-2.

Table 6-2. *Different Methods for Creating a DataFrame*

Method	Syntax

By combining Series objects	CODE:
	`student_ages=pd.Series([22,24,20]) #series 1`
	`teacher_ages=pd.Series([40,50,45])#series 2`
	`combined_ages=pd.DataFrame([student_ages,`
	`teacher_ages]) #DataFrame`
	`combined_ages.columns=['class 1','class 2',`
	`'class 3']#naming columns`
	`combined_ages`
	
	Output:
	
	class 1 **class 2** **class 3**
	0 22 24 20
	1 40 50 45
	
	Here, we are defining two Series and then using the *pd.DataFrame* function to create a new DataFrame called "combined_ages". We give names to columns in a separate step.

(*continued*)

Table 6-2. (*continued*)

Method	Syntax

From a dictionary	CODE:
	`combined_ages=pd.DataFrame({'class 1':[22,40],'class 2':[24,50],'class 3':[20,45]})`
	`combined_ages`
	
	Output:
	
	class 1 **class 2** **class 3**
	0 22 24 20
	1 40 50 45
	
	A dictionary is passed as an argument to the *pd.DataFrame* function (with the column names forming keys, and values in each column enclosed in a list).

From a numpy array	CODE:
	`numerical_df=pd.DataFrame(np.arange(1,9).reshape(2,4))`
	`numerical_df`
	
	Output:
	
	0 **1** **2** **3**
	0 1 2 3 4
	1 5 6 7 8
	
	Here, we create a NumPy array first using the *np.arange* function. Then we reshape this array into a DataFrame with two rows and four columns.

(*continued*)

Table 6-2. (*continued*)

Method	Syntax

Using a set of tuples	CODE:
	`combined_ages=pd.DataFrame([(22,24,20),(40,50,45)],columns=['class 1','class 2','class 3'])`
	`combined_ages`
	Output:
	class 1 **class 2** **class 3**
	0 22 24 20
	1 40 50 45
	We have re-created the "combined_ages" DataFrame using a set of tuples. Each tuple is equivalent to a row in a DataFrame.

To sum up, we can create a DataFrame using a dictionary, a set of tuples, and by combining Series objects. Each of these methods uses the *pd.DataFrame* function. Note that the characters "D" and "F" in this method are in uppercase; pd.dataframe does not work.

Creating DataFrames by importing data from other formats

Pandas can read data from a wide variety of formats using its reader functions (refer to the complete list of supported formats here: https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html). The following are some of the commonly used formats.

From a CSV file:

The *read_csv* function can be used to read data from a CSV file into a DataFrame, as shown in the following.

CODE:

```
titanic=pd.read_csv('titanic.csv')
```

Reading data from CSV files is one of the most common ways to create a DataFrame. CSV files are comma-separated files for storing and retrieving values, where each line is equivalent to a row. Remember to upload the CSV file in Jupyter using the upload button on the Jupyter home page (Figure 6-1), before calling the "read_csv" function.

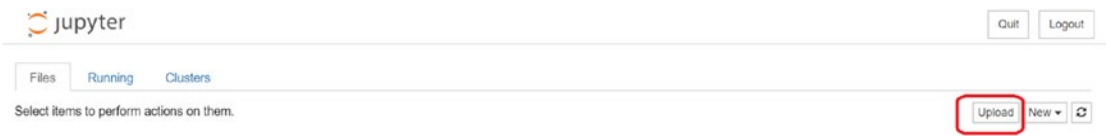

Figure 6-1. *Jupyter file upload*

From an Excel file:

Pandas provides support for importing data from both xls and xlsx file formats using the *pd.read_excel* function, as shown in the following.

CODE:

```
titanic_excel=pd.read_excel('titanic.xls')
```

From a JSON file:

JSON stands for JavaScript Object Notation and is a cross-platform file format for transmitting and exchanging data between the client and server. Pandas provides the function *read_json* to read data from a JSON file, as shown in the following.

CODE:

```
titanic=pd.read_json('titanic-json.json')
```

From an HTML file:

We can also import data from a web page using the *pd.read_html* function.

In the following example, this function parses the tables on the web page into DataFrame objects. This function returns a list of DataFrame objects which correspond to the tables on the web page. In the following example, table[0] corresponds to the first table on the mentioned URL.

CODE:

```
url="https://www.w3schools.com/sql/sql_create_table.asp"
table=pd.read_html(url)
table[0]
```

Output:

	PersonID	LastName	FirstName	Address	City
0	NaN	NaN	NaN	NaN	NaN

Further reading: See the complete list of supported formats in Pandas and the functions for reading data from such formats:

https://pandas.pydata.org/pandas-docs/stable/reference/io.html

Accessing attributes in a DataFrame

In this section, we look at how to access the attributes in a DataFrame object.

We use the following DataFrame:

CODE:

```
combined_ages=pd.DataFrame({'class 1':[22,40],'class 2':[24,50],
'class 3':[20,45]})
```

Attributes

The index attribute, when used with a DataFrame object, gives the type of an index object and its values.

CODE:

```
combined_ages.index
```

Output:

```
RangeIndex(start=0, stop=2, step=1)
```

The columns attribute gives you information about the columns (their names and data type).

CODE:

```
combined_ages.columns
```

Output:

```
Index(['class 1', 'class 2', 'class 3'], dtype="object")
```

The index object and column object are both types of index objects. While the index object has a type *RangeIndex*, the columns object has a type "Index". The values of the index object act as row labels, while those of the column object act as column labels.

Accessing the values in the DataFrame

Using the values attribute, you can obtain the data stored in the DataFrame. The output, as you can see, is an array containing the values.

CODE:

```
combined_ages.values
```

Output:

```
array([[22, 24, 20],
       [40, 50, 45]], dtype=int64)
```

Modifying DataFrame objects

In this section, we will learn how to change the names of columns and add and delete columns and rows.

Renaming columns

The names of the columns can be changed using the *rename* method. A dictionary is passed as an argument to this method. The keys for this dictionary are the old column names, and the values are the new column names.

CODE:

```
combined_ages.rename(columns={'class 1':'batch 1','class 2':'batch
2','class 3':'batch 3'},inplace=True)
combined_ages
```

Output:

	batch 1	batch 2	batch 3
0	22	24	20
1	40	50	45

The reason we use the *inplace* parameter so that the changes are made in the actual DataFrame object.

Renaming can also be done by accessing the columns attribute directly and mentioning the new column names in an array, as shown in the following example.

CODE:

```
combined_ages.columns=['batch 1','batch 2','batch 3']
```

Renaming using the dictionary format is a more straightforward method for renaming columns, and the changes are made to the original DataFrame object. The disadvantage with this method is that one needs to remember the order of the columns in the DataFrame. When we used the *rename* method, we used a dictionary where we knew which column names we were changing.

Replacing values or observations in a DataFrame

The *replace* method can be used to replace values in a DataFrame. We can again use the dictionary format, with the key/value pair representing the old and new values. Here, we replace the value 22 with the value 33.

CODE:

```
combined_ages.replace({22:33})
```

Output:

	class 1	class 2	class 3
0	33	24	20
1	40	50	45

Adding a new column to a DataFrame

There are four ways to insert a new column in a DataFrame, as shown in Table 6-3.

Table 6-3. *Adding a New Column to a DataFrame*

Method of column insertion	Syntax

With the indexing operator, []	CODE:
	`combined_ages['class 4']=[18,40]`
	`combined_ages`
	Output:
	class 1 class 2 class 3 class 4
	0 22 24 20 18
	1 40 50 45 40
	By mentioning the column name as a string within the indexing operator and assigning it values, we can add a column.

Using the *insert* method	CODE:
	`combined_ages.insert(2,'class 0',[18,35])`
	`combined_ages`
	Output:
	class 1 class 2 class 0 class 3
	0 22 24 18 20
	1 40 50 35 45
	The *insert* method can be used for adding a column. Three arguments need to be passed to this method, mentioned in the following.
	The first argument is the index where you want to insert the new column (in this case the index is 2, which means that the new column is added as the third column of our DataFrame)
	The second argument is the name of the new column you want to insert ("class 0" in this example)
	The third argument is the list containing the values of the new column (18 and 35 in this case)
	All the three parameters are mandatory for the *insert* method to be able to add a column successfully.

(*continued*)

Table 6-3. (*continued*)

Method of column insertion	Syntax

Using the *loc* indexer	CODE:
	`combined_ages.loc[:,'class 4']=[20,40]`
	`combined_ages`
	
	Output:
	
	class 1 **class 2** **class 3** **class 4**
	0 22 24 20 20
	1 40 50 45 40
	
	The *loc* indexer is generally used for retrieval of values in from Series and DataFrames, but it can also be used for inserting a column. In the preceding statement, all the rows are selected using the : operator. This operator is followed by the name of the column to be inserted. The values for this column are enclosed within a list.

Using the *concat* function	CODE:
	`class5=pd.Series([31,48])`
	`combined_ages=pd.concat([combined_ages,class5],axis=1)`
	`combined_ages`
	
	Output:
	
	class 1 **class 2** **class 3** **0**
	0 22 24 20 31
	1 40 50 45 48
	
	First, the column to be added (“class5” in this case) is defined as a Series object. It is then added to the DataFrame object using the *pd.concat* function. The axis needs to be mentioned as “1” since the new data is being added along the column axis.

In summary, we can add a column to a DataFrame using the indexing operator, *loc* indexer, *insert* method, or *concat* function. The most straightforward and commonly used method for adding a column is by using the indexing operator [].

Inserting rows in a DataFrame

There are two methods for adding rows in a DataFrame, either by using the *append* method or with the *concat* function, as shown in Table 6-4.

Table 6-4. *Adding a New Row to a DataFrame*

Method for row insertion	Syntax

Using the *append* method	CODE:
	`combined_ages=combined_ages.append({'class 1':35,'class 2':33,'class 3':21},ignore_index=True)`
	`combined_ages`
	
	Output:
	
	class 1 **class 2** **class 3**
	0 22 24 20
	1 40 50 45
	2 35 33 21
	
	The argument to the *append* method- the data that needs to be added - is defined as a dictionary. This dictionary is then passed as an argument to the *append* method. Setting the *ignore_index=True* parameter prevents an error from being thrown. This parameter resets the index. While using the *append* method, we need to ensure that we either use the *ignore_index* parameter or give a name to a Series before appending it to a DataFrame. Note that the *append* method does not have an *inplace* parameter that would ensure that the changes reflect in the original object; hence we need to set the original object to point to the new object created using append, as shown in the preceding code.

(*continued*)

Table 6-4. (*continued*)

Method for row insertion	Syntax

Using the *pd.concat* function	CODE:
	`new_row=pd.DataFrame([{'class 1':32,'class 2':37, 'class 3':41}])`
	`pd.concat([combined_ages,new_row])`
	
	Output:
	
	class 1 **class 2** **class 3**
	0 22 24 20
	1 40 50 45
	0 32 37 41
	
	The *pd.concat* function is used to add new rows as shown in the preceding syntax. The new row to be added is defined as a DataFrame object. Then the *pd.concat* function is called and the names of the two DataFrames (the original DataFrame and the new row defined as a DataFrame) are passed as arguments.

In summary, we can use either the *append* method or *concat* function for adding rows to a DataFrame.

Deleting columns from a DataFrame

Three methods can be used to delete a column from a DataFrame, as shown in Table 6-5.

Table 6-5. *Deleting a Column from a DataFrame*

Method for deletion of column	Syntax

del function	CODE:
	`del combined_ages['class 3']`
	`combined_ages`
	
	Output:
	class 1 **class 2**
	0 22 24
	1 40 50
	
	The preceding statement deletes the last column (with the name,"class 3").
	
	Note that the deletion occurs inplace, that is, in the original DataFrame itself.

Using the *pop* method	CODE:
	`combined_ages.pop('class 2')`
	
	Output:
	`0 24`
	`1 50`
	`Name: class 2, dtype: int64`
	
	The *pop* method deletes a column inplace and returns the deleted column as a Series object

(*continued*)

Table 6-5. (*continued*)

Method for deletion of column	Syntax

Using the drop method	CODE:
	`combined_ages.drop(['class 1'],axis=1,inplace=True)`
	`combined_ages`
	
	Output:
	
	|
		class 2	class 3

---	---	---
0	24	20
1	50	45	 The column(s) that needs to be dropped is mentioned as a string within a list, which is then passed as an argument to the *drop* method. Since the *drop* method removes rows (axis=0) by default, we need to specify the axis value as "1" if we want to remove a column. Unlike the *del* function and *pop* method, the deletion using the *drop* method does not occur in the original DataFrame object, and therefore, we need to add the *inplace* parameter.

To sum up, we can use the *del* function, *pop* method, or *drop* method to delete a column from a DataFrame.

Deleting a row from a DataFrame

There are two methods for removing rows from a DataFrame – either by using a Boolean selection or by using the drop method, as shown in Table 6-6.

Table 6-6. *Deleting Row from a DataFrame*

Method of row deletion	Syntax
Using a Boolean selection	CODE: `combined_ages[~(combined_ages.values<50)]` Output: **class 1** **class 2** **class 3** **1** 40 50 45 We use the NOT operator (~) to remove the rows that we do not want. Here, we remove all values in the DataFrame that are less than 50.
Using the *drop* method	CODE: `combined_ages.drop(1)` Output: **class 1** **class 2** **class 3** **0** 22 24 20 Here, we remove the second row, which has a row index of 1. If there is more than one row to be removed, we need to specify the indexes of the rows in a list.

Thus, we can use either a Boolean selection or the drop method to remove rows from a DataFrame. Since the drop method works with the removal of both rows and columns, it can be used uniformly. Remember to add the required parameters to the drop method. For removing columns, the *axis* (=1) parameter needs to be added. For changes to reflect in the original DataFrame, the *inplace* (=True) parameter needs to be included.

Indexing

Indexing is fundamental to Pandas and is what makes retrieval and access to data much faster compared to other tools. It is crucial to set an appropriate index to optimize performance. An index is implemented in NumPy as an immutable (cannot be modified)

array and contains hashable objects. A hashable object is one that can be converted to an integer value based on its contents (similar to mapping in a dictionary). Objects with different values will have different hash values.

Pandas has two types of indexes - a row index (vertical) with labels attached to rows, and a column index with labels (column names) for every column.

Let us now explore index objects – their data types, their properties, and how they speed up access to data.

Type of an index object

An index object has a data type, some of which are listed here.

- Index: This is a generic index type; the column index has this type.
- RangeIndex: Default index type in Pandas (used when an index is not defined separately), implemented as a range of increasing integers. This index type helps with saving memory.
- Int64Index: An index type containing integers as labels. For this index type, the index labels need not be equally spaced, whereas this is required for an index of type RangeIndex.
- Float64Index: Contains floating-point numbers (numbers with a decimal point) as index labels.
- IntervalIndex: Contains intervals (for instance, the interval between two integers) as labels.
- CategoricalIndex: A limited and finite set of values.
- DateTimeIndex: Used to represent date and time, like in time-series data.
- PeriodIndex: Represents periods like quarters, months, or years.
- TimedeltaIndex: Represents duration between two periods of time or two dates.
- MultiIndex: Hierarchical index with multiple levels.

Further reading:

Learn more about types of indexes here: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Index.html

Creating a custom index and using columns as indexes

When a Pandas object is created, a default index is created of the type RangeIndex, as mentioned earlier. An index of this type has the first label value as 0 (which corresponds to the first item of the Pandas Series or DataFrame), and the second label as 1, following an arithmetic progression with a spacing of one integer.

We can set a customized index, using either the index parameter or attribute. In the Series and DataFrame objects we created earlier, we were just setting values for the individual items, and in the absence of labels for the index object, the default index (of type RangeIndex) was used.

We can use the index parameter when we define a Series or DataFrame to give custom values to the index labels.

CODE:

```
periodic_table=pd.DataFrame({'Element':['Hydrogen','Helium','Lithium',
'Beryllium','Boron']},index=['H','He','Li','Be','B'])
```

Output:

	Element
H	Hydrogen
He	Helium
Li	Lithium
Be	Beryllium
B	Boron

If we skip the index parameter during the creation of the object, we can set the labels using the index attribute, as shown here.

CODE:

```
periodic_table.index=['H','He','Li','Be','B']
```

The *set_index* method can be used to set an index using an existing column, as demonstrated in the following:

CODE:

```
periodic_table=pd.DataFrame({'Element':['Hydrogen','Helium','Lithium',
'Beryllium','Boron'],'Symbols':['H','He','Li','Be','B']})
periodic_table.set_index(['Symbols'])
```

Output:

	Element
Symbols	
H	Hydrogen
He	Helium
Li	Lithium
Be	Beryllium
B	Boron

The index can be made a column again or reset using the reset_index method:

CODE:

```
periodic_table.reset_index()
```

Output:

	index	Element	Symbols
0	0	Hydrogen	H
1	1	Helium	He
2	2	Lithium	Li
3	3	Beryllium	Be
4	4	Boron	B

We can also set the index when we read data from an external file into a DataFrame, using the *index_col* parameter, as shown in the following.

CODE:

```
titanic=pd.read_csv('titanic.csv',index_col='PassengerId')
titanic.head()
```

Output:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Indexes and speed of data retrieval

We know that indexes dramatically improve the speed of access to data. Let us understand this with the help of an example.

Consider the following DataFrame:

CODE:

```
periodic_table=pd.DataFrame({'Atomic Number':[1,2,3,4,5],'Element':
['Hydrogen','Helium','Lithium','Beryllium','Boron'],'Symbol':['H','He',
'Li','Be','B']})
```

Output:

	Atomic Number	Element	Symbol
0	1	Hydrogen	H
1	2	Helium	He
2	3	Lithium	Li
3	4	Beryllium	Be
4	5	Boron	B

Searching without using an index

Now, try retrieving the element with atomic number 2 without the use of an index and measure the time taken for retrieval using the *timeit* magic function. When the index is not used, a linear search is performed to retrieve an element, which is relatively time consuming.

CODE:

```
%timeit periodic_table[periodic_table['Atomic Number']==2]
```

Output:

```
1.66 ms ± 99.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Search using an index

Now, set the "Atomic Number" column as the index and use the *loc* indexer to see how much time the search takes now:

CODE:

```
new_periodic_table=periodic_table.set_index(['Atomic Number'])
%timeit new_periodic_table.loc[2]
```

Output:

```
281 µs ± 14.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The search operation, when performed without using an index, was of the order of milliseconds (around 1.66 ms). With the use of indexes, the time taken for the retrieval operation is now of the order of microseconds (281 µs), which is a significant improvement.

Immutability of an index

As mentioned earlier, the index object is immutable - once defined, the index object or its labels cannot be modified.

As an example, let us try changing one of the index labels in the periodic table DataFrame we just defined, as shown in the following. We get an error in the output since we are trying to operate on an immutable object.

CODE:

```
periodic_table.index[2]=0
```

Output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-cd2fece917cb> in <module>
----> 1periodic_table.index[2]=0

~\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__
(self, key, value)
   3936
   3937def __setitem__(self, key, value):
-> 3938raiseTypeError("Index does not support mutable operations")
   3939
   3940def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

While the values of an Index object cannot be changed, we can retrieve information about the index using its attributes, like the values contained in the Index object, whether there are any null values, and so on.

Let us look at some of the index attributes with some examples:

Considering the column index in the following DataFrame:

CODE:

```
periodic_table=pd.DataFrame({'Element':['Hydrogen','Helium','Lithium','Bery
llium','Boron']},index=['H','He','Li','Be','B'])

column_index=periodic_table.columns
```

Some of the attributes of the column index are

1.values attribute: Returns the column names

CODE:

```
column_index.values
```

Output:

```
array(['Element'], dtype=object)
```

2.hasnans attribute: Returns a Boolean True or False value based on the presence of null values.

CODE:

```
column_index.hasnans
```

Output:

```
False
```

3.nbytes attribute: Returns the number of bytes occupied in memory

CODE:

```
column_index.nbytes
```

Output:

```
8
```

Further reading: For a complete list of attributes, refer to the following documentation: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Index.html

Alignment of indexes

When two Pandas objects are added, their index labels are checked for alignment. For items that have matching indexes, their values are added or concatenated. Where the indexes do not match, the value corresponding to that index in the resultant object is null (*np.NaN*).

Let us understand this with an example. Here, we see that the 0 index label in s1 does not have a match in s2, and the last index label (10) in s2 does not have a match in s1. These values equal null when the objects are combined. All other values, where the index labels align, are added together.

CODE:

```
s1=pd.Series(np.arange(10),index=np.arange(10))
s2=pd.Series(np.arange(10),index=np.arange(1,11))
s1+s2
```

Output:

```
0      NaN
1      1.0
2      3.0
3      5.0
4      7.0
5      9.0
6     11.0
7     13.0
8     15.0
9     17.0
10     NaN
dtype: float64
```

Set operations on indexes

We can perform set operations like union, difference, and symmetric difference on indexes from different objects.

Consider the following indexes, "i1" and "i2", created from two Series objects ("s1" and "s2") we created in the previous section:

CODE:

```
i1=s1.index
i2=s2.index
```

Union operation

All elements present in both sets are returned.

CODE:

```
i1.union(i2)
```

Output:

```
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype="int64")
```

Difference operation

Elements present in one set, but not in the other, are returned.

CODE:

```
i1.difference(i2) #elements present in i1 but not in i2
```

Output:

```
Int64Index([0], dtype="int64")
```

Symmetric difference operation

Elements not common to the two sets are returned. This operation differs from the Difference operation in that it takes into the uncommon elements in both sets:

CODE:

```
i1.symmetric_difference(i2)
```

Output:

```
Int64Index([0, 10], dtype="int64")
```

You can also perform arithmetic operations on two index objects, as shown in the following.

CODE:

```
i1-i2
```

Output:

```
Int64Index([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1], dtype="int64")
```

Data types in Pandas

The data types used in Pandas are derived from NumPy, except for the "category" data type for qualitative data, which is defined in Pandas. The common data types include

- object (for storing mixed data like numbers, strings, etc.)
- int64 (for integer values)
- float64 (for numbers with decimal points)

- Datetime (for storing date and time data)
- Category (for variables containing only a few distinct values, like 'True'/'False', or some limited ordered categories like 'one'/'two'/'three'/'four')

Obtaining information about data types

We now understand how to retrieve information about the data types of columns.

Import the *subset-covid-data.csv* file and read the data into a DataFrame, as shown in the following.

CODE:

```
data=pd.read_csv('subset-covid-data.csv')
data.head()
```

Output:

	dateRep	day	month	year	cases	deaths	countriesAndTerritories	geoId	countryterritoryCode	popData2018	continentExp
0	2020-05-27	27	5	2020	658	1	Afghanistan	AF	AFG	37172386.0	Asia
1	2020-05-26	26	5	2020	591	1	Afghanistan	AF	AFG	37172386.0	Asia
2	2020-05-25	25	5	2020	584	2	Afghanistan	AF	AFG	37172386.0	Asia
3	2020-05-24	24	5	2020	782	11	Afghanistan	AF	AFG	37172386.0	Asia
4	2020-05-23	23	5	2020	540	12	Afghanistan	AF	AFG	37172386.0	Asia

Using the *dtypes* attribute, we can obtain the type of columns in this DataFrame.

CODE:

```
data.dtypes
```

Output:

```
country          object
continent        object
date             object
```

```
day                int64
month              int64
year               int64
cases              int64
deaths             int64
country_code      object
population       float64
dtype: object
```

As we discussed in the previous chapter, the kind of mathematical operations and graphs that can be used differ for categorical and continuous variables. Knowing the data types of columns helps us figure out how to analyze the variables. The columns that have the Pandas data type "object" or "category" are categorical variables, whereas variables with data types like "int64" and "float64" are continuous.

Get the count of each data type

To obtain the number of columns belonging to each data type, we use the *get_dtype_counts* method:

CODE:

```
data.get_dtype_counts()
```

Output:

```
float64    1
int64      5
object     4
dtype: int64
```

Select particular data types

Using the *select_dtypes* method, we can filter the columns based on the type of data you want to select:

CODE:

```
data.select_dtypes(include='number').head()
```

```
#This will select all columns that have integer and floating-point data and exclude the rest. The head parameter has been used to limit the number of records being displayed.
```

Output:

	day	month	year	cases	deaths	population
0	12	4	2020	34	3	37172386.0
1	12	4	2020	17	0	2866376.0
2	12	4	2020	64	19	42228429.0
3	12	4	2020	21	2	77006.0
4	12	4	2020	0	0	30809762.0

Calculating the memory usage and changing data types of columns

We can find the memory usage (in bytes) of a Series or a DataFrame by using the *memory_usage* method. We include the *deep* parameter while using this method to get a more comprehensive picture of the memory usage at the system level.

CODE:

```
data['continent'].memory_usage(deep=True)
```

Output:

```
13030
```

Let us see if we can reduce the memory usage of this column. First, let us find its current data type.

CODE:

```
data['continent'].dtype
```

Output:

```
dtype('O')
```

As we can see, this column occupies 13030 bytes of memory and has a data type of "O". The Pandas categorical data type is useful for storing qualitative variables that have only a few unique values, as this reduces memory usage. Since the continent column has only a few unique values ("Europe", "Asia", "America", "Africa", "Oceania"), let us change the data type of this column from *object* to *categorical*, and see if this reduces memory usage. We use the *astype* method for changing data types.

CODE:

```
data['continent']=data['continent'].astype('category')
data['continent'].memory_usage(deep=True)
```

Output:

```
823
```

The memory usage seems to have reduced quite a bit after changing the data type. Let us calculate the exact percentage reduction.

CODE:

```
(13030-823)/13030
```

Output:

```
0.936838066001535
```

A significant reduction in memory usage, around 93%, has been achieved by changing the data type from object to categorical.

Indexers and selection of subsets of data

In Pandas, there are many ways of selecting and accessing data, as listed in the following.

- *loc* and *iloc* indexers
- *ix* indexer
- *at* and *iat* indexers
- indexing operator []

The preferred method for data retrieval is through the use of the *loc* and *iloc* indexers. Both indexers and the indexing operator enable access to an object using indexes. Note that an indexer is different from the indexing operator, which is a pair of square brackets containing the index. While we have used the indexing operator [], for selecting data from objects like lists, tuples, and NumPy, the use of this operator is not recommended.

For instance, if we want to select the first row in Pandas, we would use the first statement given in the following.

CODE:

```
data.iloc[0] #correct
data[0] #incorrect
```

Understanding loc and iloc indexers

The **loc indexer** works by selecting data using index labels, which is similar to how data is selected in dictionaries in Python, using keys associated with values.

The **iloc indexer,** on the other hand, selects data using the integer location, which is similar to how individual elements are in lists and arrays.

Note that *loc* and *iloc* being indexers, and are followed by square brackets, not round brackets (like in the case of functions or methods). The index values before the comma refer to the row indexes, and the index values after the comma refer to the column indexes.

Let us consider some examples to understand how the *loc* and *iloc* indexers work. We again use the covid-19 dataset ("subset-covid-data.csv") for these examples.

CODE:

```
data=pd.read_csv('subset-covid-data.csv',index_col='date')
```

Here, we are using the column 'date' as the index.

Selecting consecutive rows

We can use *iloc* for this, since we know the index (first five) of the rows to be retrieved:

CODE:

```
data.iloc[0:5]
```

Output:

date	country	continent	day	month	year	cases	deaths	country_code	population
2020-04-12	Afghanistan	Asia	12	4	2020	34	3	AFG	37172386.0
2020-04-12	Albania	Europe	12	4	2020	17	0	ALB	2866376.0
2020-04-12	Algeria	Africa	12	4	2020	64	19	DZA	42228429.0
2020-04-12	Andorra	Europe	12	4	2020	21	2	AND	77006.0
2020-04-12	Angola	Africa	12	4	2020	0	0	AGO	30809762.0

Note that we mention only the row indexes, and in the absence of a column index, all the columns are selected by default.

Selecting consecutive columns

We can use *iloc* for this since the index values (0,1,2) for the first three columns are known.

CODE:

```
data.iloc[:,:3]
```

Or

```
data.iloc[:,0:3]
```

Output (only first five rows shown)

date	country	continent	day
2020-04-12	Afghanistan	Asia	12
2020-04-12	Albania	Europe	12
2020-04-12	Algeria	Africa	12
2020-04-12	Andorra	Europe	12
2020-04-12	Angola	Africa	12

While we can skip the column indexes, we cannot skip the row indexes.

The following syntax would not work:

CODE:

```
data.iloc[,0:3] #incorrect
```

In this example, we are selecting all the rows and three columns. On either side of the colon (:) symbol, we have a start and a stop value. If both start and stop values are missing, it means all values are to be selected. If the starting index is missing, it assumes a default value of 0. If the stop index value is missing, it assumes the last possible positional value of the index (one minus the number of columns or rows).

Selecting a single row

Let us select the 100th row using the *iloc* indexer. The 100th row has an index of 99 (since the index numbering starts from 0).

CODE:

```
data.iloc[99]
```

Output:

```
country             Jamaica
continent           America
day                      12
month                     4
year                   2020
cases                     4
deaths                    0
country_code            JAM
population      2.93486e+06
Name: 2020-04-12, dtype: object
```

Selecting rows using their index labels

Select the rows with the date as 2020-04-12. Here, we use the loc indexer since we know the index labels for the rows that need to be selected but do not know their position.

```
data.loc['2020-04-12']
```

Output (only first five rows shown):

date	country	continent	day	month	year	cases	deaths	country_code	population
2020-04-12	Afghanistan	Asia	12	4	2020	34	3	AFG	37172386.0
2020-04-12	Albania	Europe	12	4	2020	17	0	ALB	2866376.0
2020-04-12	Algeria	Africa	12	4	2020	64	19	DZA	42228429.0
2020-04-12	Andorra	Europe	12	4	2020	21	2	AND	77006.0
2020-04-12	Angola	Africa	12	4	2020	0	0	AGO	30809762.0

Selecting columns using their name

Let us select the column named "cases". Since the name of a column acts as its index label, we can use the loc indexer.

CODE:

```
data.loc[:,'cases']
```

Output:

```
date
2020-04-12       34
2020-04-12       17
2020-04-12       64
2020-04-12       21
2020-04-12        0
```

Using negative index values for selection

Let us select the first five rows and last three columns. Here, we are using negative indices to select the last three columns. The last column has a positional index value of –1, the last but one column has an index value of –2, and so on. The step size is –1. We skip the start value for the row slice (:5) since the default value is 0.

CODE:

```
data.iloc[:5,-1:-4:-1]
```

Output:

date	population	country_code	deaths
2020-04-12	37172386.0	AFG	3
2020-04-12	2866376.0	ALB	0
2020-04-12	42228429.0	DZA	19
2020-04-12	77006.0	AND	2
2020-04-12	30809762.0	AGO	0

Selecting nonconsecutive rows and columns

To select a set of rows or columns that are not consecutive, we need to enclose the rows or column index positions or labels in a list. In the following example, we select the second and fifth row, along with the first and fourth columns.

CODE:

```
data.iloc[[1,5],[0,3]]
```

Output:

date	country	month
2020-04-12	Albania	4
2020-04-12	Anguilla	4

Other (less commonly used) indexers for data access

The reason the indexers *loc* and *iloc* are recommended for slicing or selecting subsets of data from Series and DataFrame objects is that they have clear rules to select data either exclusively by their labels (in case of loc) or through their position (in case of iloc). However, it is essential to understand the other indexers supported in Pandas, explained in the following section.

ix indexer

The *ix* indexer allows us to select data by combining the index label and the location. This method of selection is in contrast to the method used by the *loc* and *iloc* indexers, which do not allow us to mix up the position and the label. With the *ix* indexer not having a clear rule to select data, there is room for much ambiguity, and this indexer has now been deprecated (which means that although it is still supported, it should not be used). For demonstration purposes, let us see how the ix indexer works. Let us select the first five rows of the column `cases' in our dataset.

CODE:

```
data.ix[:5,'cases']
```

Output:

```
 C:\Users\RA\Anaconda3\lib\site-packages\ipykernel_launcher.py:1:
DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-
deprecated
  """Entry point for launching an IPython kernel.

date
2020-04-12    34
2020-04-12    17
2020-04-12    64
```

```
2020-04-12    21
2020-04-12     0
Name: cases, dtype: int64
```

Note that the use of the ix indicator leads to a warning asking the user to use either loc or iloc in its place.

The indexing operator - [ ]

Even though the indexing operator is not the preferred mode for data selection or slicing in Pandas, it still has its uses. One appropriate application of this operator is for selecting columns from a DataFrame. The argument is the name of the column that is mentioned as a string (enclosed within quotes).

For instance, the following statement would select the population column from our COVID dataset.

CODE:

```
data['population']
```

Output (only first five rows shown):

```
date
2020-04-12     37172386.0
2020-04-12      2866376.0
2020-04-12     42228429.0
2020-04-12        77006.0
2020-04-12     30809762.0
```

To select multiple columns, we pass the column names as strings within a list, as shown in the example in the following:

CODE:

```
data[['country','population']]
```

Output (truncated):

	countriesAndTerritories	popData2018
0	Afghanistan	37172386.0
1	Afghanistan	37172386.0
2	Afghanistan	37172386.0
3	Afghanistan	37172386.0
4	Afghanistan	37172386.0

The indexing operator can also be used for selecting a set of consecutive rows.

CODE:

```
data[:3]
```

Output:

dateRep	day	month	year	cases	deaths	countriesAndTerritories	geoId	countryterritoryCode	popData2018	continentExp
2020-05-27	27	5	2020	658	1	Afghanistan	AF	AFG	37172386.0	Asia
2020-05-26	26	5	2020	591	1	Afghanistan	AF	AFG	37172386.0	Asia
2020-05-25	25	5	2020	584	2	Afghanistan	AF	AFG	37172386.0	Asia

However, it cannot be used to select a series of nonconsecutive rows, as this will raise an error. The following statement would not work.

CODE:

```
data[[3,5]] #incorrect
```

Another limitation of the indexing operator is that it cannot be used to select rows and columns simultaneously. The following statement would also not work.

CODE:

```
data[:,3] #incorrect
```

at and iat indexers

There are two other less commonly used indexers – *at* (similar to *loc*, works with labels) and *iat* (similar to *iloc*, works with positions). The three main features of the *at* and *iat* indexers are

- They can be used only for selecting scalar (single) values from a Series or DataFrame.
- Both row and column indexes need to be supplied as arguments to these indexers since they return a single value. We cannot obtain a set of rows or columns with this indexer, which is possible with the other indexers.
- These indexers are quicker at retrieving data than loc and iloc.

Let us understand how these indexers work with the help of an example.

Import the subset-covid-data.csv dataset.

```
data=pd.read_csv('subset-covid-data.csv')
```

The *at* indexer works just like loc, and you need to pass the row index label and the column name as arguments.

Let us try to retrieve the population value in the first row. Since we have not set an index for this DataFrame, the index labels and positions would be the same.

CODE:

```
data.at[0,'population']
#0 is the index label as well as the position
```

Output:

```
37172386.0
```

The *iat* indexer is similar to the *iloc* indexer, with the row/column indexes being passed as arguments.

CODE:

```
data.iat[0,9]
#0,9 is the position of the first record of the population column
```

The output is the same as the one for the previous statement.

Boolean indexing for selecting subsets of data

In the previous examples that we looked at, we used various indexers to retrieve data based on the position or label. With Boolean indexing, we use conditional statements to filter data based on their values. A single condition may be specified, or multiple conditions can be combined using the bitwise operators - & (and), | (or), ~ (not).

Let us consider an example to understand this. Here, we select all records where the name of the continent is "Asia", and the country name starts with the letter "C".

CODE:

```
data[(data['continent']=='Asia') & (data['country'].str.startswith('C'))]
```

Output:

	country	continent	date	day	month	year	cases	deaths	country_code	population
33	Cambodia	Asia	2020-04-12	12	4	2020	2	0	KHM	1.624980e+07
42	China	Asia	2020-04-12	12	4	2020	93	0	CHN	1.392730e+09

Using the query method to retrieve data

While we combine multiple conditions as in the previous example, the readability of the code may suffer. The *query* method can be used in such cases.

Let us retrieve all the records where the name of the continent is "Asia" and the number of cases is higher than 500. Note the syntax where we enclose each condition within double quotes and use the logical *and* operator, instead of the bitwise operator, &.

CODE:

```
data.query("(continent=='Asia')""and (cases>=500)")
```

Output:

	country	continent	date	day	month	year	cases	deaths	country_code	population
91	India	Asia	2020-04-12	12	4	2020	909	34	IND	1.352617e+09
93	Iran	Asia	2020-04-12	12	4	2020	1837	125	IRN	8.180027e+07
100	Japan	Asia	2020-04-12	12	4	2020	1401	10	JPN	1.265291e+08
190	Turkey	Asia	2020-04-12	12	4	2020	5138	95	TUR	8.231972e+07

Further reading

See more on:

- Query method: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html
- Indexing in Pandas: https://pandas.pydata.org/docs/user_guide/indexing.html

Operators in Pandas

Pandas uses the following operators that can be applied to a whole series. While Python would require a loop to iterate through every element in a list or dictionary, Pandas takes advantage of the feature of vectorization implemented in NumPy that enables these operators to be applied on every element in a sequence, eliminating the need for iteration and loops. The different types of operators are listed in Table 6-7.

Table 6-7. *Pandas Operators*

Type of operator	Operators included
Arithmetic Operators	+addition), -(subtraction), *(multiplication),**(power),%(remainder operator),/(division),//(floor division, for getting the quotient). The functions performed by arithmetic operators can be replicated using the following methods: add for +, sub for -, mul for *, div for /, mod for %, and pow for **.
Comparison Operators	== (equality),<(less than),>(greater than),<=(less than or equal to),>=(greater than or equal to),!=(not equal to)
Logical Operators	&,|,~. Pandas, like NumPy, uses the bitwise operators (&,|,~) as logical operators, as these operators operate on every element of a Series. Note that these operators are different from the logical operators used in Python, where the keywords *and*, *or*, and *not* are used.

Representing dates and times in Pandas

In Pandas, there is a single Timestamp function that can be used to define a date, time, or a combination of a date and a time. This is in contrast to the implementation in Python, which requires separate objects to define a date or time. The *pd.Timestamp* function is equivalent to the following functions in Python: *datetime.date*, *datetime.time*, *datetime.datetime.*

As an example, let us represent the date 25th December 2000 in Pandas using the *pd.Timestamp* function.

CODE:

```
pd.Timestamp('25/12/2000')
```

Output:

```
Timestamp('2000-12-25 00:00:00')
```

The *Timestamp* function is very flexible and accepts parameters in a variety of formats. The preceding output can also be replicated using any of the following statements.

```
#different input formats for creating a Timestamp object

pd.Timestamp('25 December 2000')

pd.Timestamp('December 25 2000')

pd.Timestamp('12/25/2000')

pd.Timestamp('25-12-2000')

pd.Timestamp(year=2000,month=12,day=25)

pd.Timestamp('25-12-2000 12 PM')

pd.Timestamp('25-12-2000 0:0.0')
```

The *pd.Timestamp* function helps us define a date, time, and a combination of these two. However, this function does not work if we need to define a duration of time. A separate function, *pd.Timedelta*, helps us create objects that store a time duration. This is equivalent to the *datetime.timedelta* function in Python.

Let us define a duration of time in Pandas using the *Timedelta* function.

CODE:

```
pd.Timedelta('45 days 9 minutes')
```

Output:

```
Timedelta('45 days 00:09:00')
```

Like the *Timestamp* function, the *Timedelta* function is flexible in what it accepts as input parameters. The preceding statement can also be written as follows.

CODE:

```
pd.Timedelta(days=45,minutes=9)
```

We can also add the *unit* parameter to create a *Timedelta* object. In the following line of code, the parameter unit with the value 'm' denotes minutes, and we add 500 minutes to the base time of 00:00:00 hours.

CODE:

```
pd.Timedelta(500,unit='s')
```

Output:

```
Timedelta('0 days 08:20:00')
```

Converting strings into Pandas Timestamp objects

Dates are generally represented as strings and need to be converted to a type that can be understood by Pandas. The *pd.to_datetime* function converts the date to a Timestamp object. Converting it to this format helps with comparing two dates, adding or subtracting a time duration from a given date, and extracting individual components (like day, month, and year) from a given date. It also helps with representing dates that are not in the traditional "day-month-year" or "month-day-year" format.

Let us consider an example to understand this. Consider the date represented as a string "11:20 AM, 2nd April 1990". We can convert this into a Timestamp object and specify the format parameter so that the individual components like the day, month, and year are parsed correctly. The *format* parameter in the *pd.to_datetime* function with its formatting codes (like %H, %M), helps with specifying the format in which this date is written. %H represents the hour, %M represents the minutes, %d is for the day, %m is for the month, and %Y is for the year.

CODE:

```
a=pd.to_datetime('11:20,02/04/1990', format='%H:%M,%d/%m/%Y')
a
```

Output:

```
Timestamp('1990-04-02 11:20:00')
```

Now that this date has been converted into a *Timestamp* object, we can perform operations on it. A *Timedelta* object can be added to a Timestamp object.

Let us add four days to this date:

CODE:

```
a+pd.Timedelta(4,unit='d')
```

Output:

```
Timestamp('1990-04-06 11:20:00')
```

Extracting the components of a Timestamp object

Once the date is converted to a Pandas Timestamp object using the *pd.to_datetime* function, the individual components of the date variable can be extracted using the relevant attributes.

CODE:

```
#extracting the month
a.month
```

Output:

```
4
```

CODE:

```
#extracting the year
a.year
```

Output:

```
1990
196
```

CODE:

```
#extracting the day
a.day
```

Output:

```
2
```

We can also use the minute and hour attribute to extract the minutes and hour from the date.

Further reading

Learn more about the Pandas Timestamp function: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Timestamp.html

Grouping and aggregation

Aggregation is the process of summarizing a group of values into a single value.

Hadley Wickham, a statistician, laid down the "Split-Apply-Combine" methodology (the paper can be accessed here: https://www.jstatsoft.org/article/view/v040i01/v40i01.pdf), which has three steps:

1. Split the data into smaller groups that are manageable and independent of each other. This is done using the *groupby* method in Pandas.
2. Apply functions on each of these groups. We can apply any of the aggregation functions, including minimum, maximum, median, mean, sum, count, standard deviation, variance, and size. Each of these aggregate functions calculate the aggregate value of the entire group. Note that we can also write a customized aggregation function.
3. Combine the results after applying functions to each group into a single combined object.

In the following section, we look at the *groupby* method, aggregation functions, the *transform*, *filter*, and *apply* methods, and the properties of the *groupby* object.

Here, we again use the same COVID-19 dataset, which shows the number of cases and deaths for all countries on 12th April 2020.

CODE:

```
df=pd.read_csv('subset-covid-data.csv')
df.head()
```

Output:

	country	continent	date	day	month	year	cases	deaths	country_code	population
0	Afghanistan	Asia	2020-04-12	12	4	2020	34	3	AFG	37172386.0
1	Albania	Europe	2020-04-12	12	4	2020	17	0	ALB	2866376.0
2	Algeria	Africa	2020-04-12	12	4	2020	64	19	DZA	42228429.0
3	Andorra	Europe	2020-04-12	12	4	2020	21	2	AND	77006.0
4	Angola	Africa	2020-04-12	12	4	2020	0	0	AGO	30809762.0

As we can see, there are several countries belonging to the same continent. Let us find the total number of cases and deaths for each continent. For this, we need to do grouping using the 'continent' column.

CODE:

```
df.groupby('continent')['cases','deaths'].sum()
```

Output:

continent	cases	deaths
Africa	714	52
America	33519	2111
Asia	12979	383
Europe	34141	3571
Oceania	68	4
Other	0	0

Here, we are grouping by the column "continent", which becomes the *grouping column*. We are aggregating the values of the number of cases and deaths, which makes the columns named "cases" and "deaths" the *aggregating columns*. The sum method, which becomes our *aggregating function*, calculates the total of cases and deaths for all countries belonging to a given continent. Whenever you perform a *groupby* operation, it is recommended that these three elements (grouping column, aggregating column, and aggregating function) be identified at the outset.

The following thirteen aggregate functions can be applied to groups: *sum()*, *max()*, *min()*, *std()*, *var()*, *mean()*, *count()*, *size()*, *sem()*, *first()*, *last()*, *describe()*, *nth()*.

We can also use the *agg* method, with *np.sum* as an attribute, which produces the same output as the previous statement:

CODE:

```
df.groupby('continent')['cases','deaths'].agg(np.sum)
```

The *agg* method can accept any of the aggregating methods, like mean, sum, max, and so on, and these methods are implemented in NumPy.

We can also pass the aggregating column and the aggregating method as a dictionary to the *agg* method, as follows, which would again produce the same output.

CODE:

```
df.groupby('continent').agg({'cases':np.sum,'deaths':np.sum})
```

If there is more than one grouping column, use a list to save the column names as strings and pass this list as an argument to the *groupby* method.

Further reading on aggregate functions: https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html#aggregation

Examining the properties of the groupby object

The result of applying the *groupby* method is a groupby object. This *groupby* object has several properties that are explained in this section.

Data type of *groupby* object

The data type of a groupby object can be accessed using the type function.

CODE:

```
grouped_continents=df.groupby('continent')
type(grouped_continents)
```

Output:

```
pandas.core.groupby.generic.DataFrameGroupBy
```

Each group of the groupby object is a separate DataFrame.

Obtaining the names of the groups

The *groupby* object has an attribute called *groups*. Using this attribute on the *groupby* object would return a dictionary, with the keys of this dictionary being the names of the groups.

CODE:

```
grouped_continents.groups.keys()
```

Output:

```
dict_keys(['Africa', 'America', 'Asia', 'Europe', 'Oceania', 'Other'])
```

Returning records with the same position in each group using the nth method

Let us say that you want to see the details of the fourth country belonging to each continent. Using the *nth* method, we can retrieve this data by using a positional index value of 3 for the fourth position.

CODE:

```
grouped_continents.nth(3)
```

Output:

continent	country	date	day	month	year	cases	deaths	country_code	population
Africa	Botswana	2020-04-12	12	4	2020	0	0	BWA	2254126.0
America	Aruba	2020-04-12	12	4	2020	6	0	ABW	105845.0
Asia	Bhutan	2020-04-12	12	4	2020	0	0	BTN	754394.0
Europe	Austria	2020-04-12	12	4	2020	247	18	AUT	8847037.0
Oceania	Guam	2020-04-12	12	4	2020	3	1	GUM	165768.0

Get all the data for a particular group using the get_group method

Use the *get_group* method with the name of the group as an argument to this method. In this example, we retrieve all data for the group named 'Europe'.

CODE:

```
grouped_continents.get_group('Europe')
```

Output (contains 54 records; only first four records shown in the following):

	country	date	day	month	year	cases	deaths	country_code	population
1	Albania	2020-04-12	12	4	2020	17	0	ALB	2866376.0
3	Andorra	2020-04-12	12	4	2020	21	2	AND	77006.0
8	Armenia	2020-04-12	12	4	2020	30	2	ARM	2951776.0
11	Austria	2020-04-12	12	4	2020	247	18	AUT	8847037.0

We have seen how to apply aggregate functions to the groupby object. Now let us look at some other functions, like filter, apply, and transform, that can also be used with a groupby object.

Filtering groups

The *filter* method removes or filters out groups based on a particular condition. While the agg (aggregate) method returns one value for each group, the filter method returns records from each group depending on whether the condition is satisfied.

Let us consider an example to understand this. We want to return all the rows for the continents where the average death rate is greater than 40. The *filter* method is called on a groupby object and the argument to the *filter* method is a lambda function or a predefined function. The *lambda* function here calculates the average death rate for every group, represented by the argument "x". This argument is a DataFrame representing each group (which is the continent in our example). If the condition is satisfied for the group, all its rows are returned. Otherwise, all the rows of the group are excluded.

CODE:

```
grouped_continents=df.groupby('continent')
grouped_continents.filter(lambda x:x['deaths'].mean()>=40)
```

Output (only first five rows shown):

	country	continent	date	day	month	year	cases	deaths	country_code	population
1	Albania	Europe	2020-04-12	12	4	2020	17	0	ALB	2866376.0
3	Andorra	Europe	2020-04-12	12	4	2020	21	2	AND	77006.0
5	Anguilla	America	2020-04-12	12	4	2020	0	0	NaN	NaN
6	Antigua_and_Barbuda	America	2020-04-12	12	4	2020	0	0	ATG	96286.0
7	Argentina	America	2020-04-12	12	4	2020	162	7	ARG	44494502.0
8	Armenia	Europe	2020-04-12	12	4	2020	30	2	ARM	2951776.0

In the output, we see that only the rows for the groups (continents) 'America' and 'Europe' are returned since these are the only groups that satisfy the condition (group mean death rate greater than 40).

Transform method and groupby

The *transform* method is another method that can be used with the *groupby* object, which applies a function on each value of the group. It returns an object that has the same rows as the original data frame or Series and is similarly indexed as well.

Let us use the *transform* method on the population column to obtain the population in millions by dividing each value in the row by 1000000.

CODE:

```
grouped_continents['population'].transform(lambda x:x/1000000)
```

Output (only first five rows and last two rows shown; actual output contains 206 rows):

```
0       37.172386
1        2.866376
2       42.228429
3        0.077006
4       30.809762
.
..
...
204     17.351822
205     14.439018
Name: population, Length: 206, dtype: float64
```

Notice that while the *filter* method returns lesser records as compared to its input object, the *transform* method returns the same number of records as the input object.

In the preceding example, we have applied the *transform* method on a Series. We can also use it on an entire DataFrame. A common application of the *transform* method is used to fill null values. Let us fill the missing values in our DataFrame with the value 0. In the output, notice that the values for the country code and population for the country 'Anguilla' (which were missing earlier) are now replaced with the value 0.

CODE:

```
grouped_continents.transform(lambda x:x.fillna(0))
```

Output:

	country	date	day	month	year	cases	deaths	country_code	population
0	Afghanistan	2020-04-12	12	4	2020	34	3	AFG	37172386.0
1	Albania	2020-04-12	12	4	2020	17	0	ALB	2866376.0
2	Algeria	2020-04-12	12	4	2020	64	19	DZA	42228429.0
3	Andorra	2020-04-12	12	4	2020	21	2	AND	77006.0
4	Angola	2020-04-12	12	4	2020	0	0	AGO	30809762.0
5	Anguilla	2020-04-12	12	4	2020	0	0	0	0.0

The *transform* method can be used with any Series or a DataFrame and not just with *groupby* objects. Creating a new column from an existing column is a common application of the *transform* method.

Apply method and groupby

The *apply* method "applies" a function to each group of the *groupby* object. The difference between the apply and transform method is that the apply method is more flexible in that it can return an object of any shape while the transform method needs to return an object of the same shape.

The *apply* method can return a single (scalar) value, Series or DataFrame, and the output need not be in the same structure as the input. Also, while the *transform* method applies the function on each column of a group, the *apply* method applies the function on the entire group.

Let us use the *apply* method to calculate the total missing values in each group (continent).

CODE:

```
grouped_continents.apply(lambda x:x.isna().sum())
```

Output:

continent	country	continent	date	day	month	year	cases	deaths	country_code	population
Africa	0	0	0	0	0	0	0	0	0	1
America	0	0	0	0	0	0	0	0	3	3
Asia	0	0	0	0	0	0	0	0	0	0
Europe	0	0	0	0	0	0	0	0	0	0
Oceania	0	0	0	0	0	0	0	0	0	0
Other	0	0	0	0	0	0	0	0	1	0

The *apply* method, similar to the transform method, can be used with Series and DataFrame objects in addition to the *groupby* object.

How to combine objects in Pandas

In Pandas, there are various functions to combine two or more objects, depending on whether we want to combine them horizontally or vertically. In this section, we cover the four methods used for combining objects - *append*, *join*, *concat*, and *merge*.

Append method for adding rows

This method is used to add rows to an existing DataFrame or Series object, but cannot be used to add columns. Let us look at this with an example:

Let us create the following DataFrame:

CODE:

```
periodic_table=pd.DataFrame({'Atomic Number':[1,2,3,4,5],'Element':
['Hydrogen','Helium','Lithium','Beryllium','Boron'],'Symbol':['H','He',
'Li','Be','B']})
```

We now add a new row (in the form of a dictionary object) by passing it as an argument to the *append* method.

We also need to remember to set the value of the *ignore_index* parameter as True. Setting it to "True" replaces the old index with a new index.

CODE:

```
periodic_table.append({'Atomic Number':6,'Element':'Carbon','Symbol':'C'},i
gnore_index=True)
```

Output:

	Atomic Number	Element	Symbol
0	1	Hydrogen	H
1	2	Helium	He
2	3	Lithium	Li
3	4	Beryllium	Be
4	5	Boron	B
5	6	Carbon	C

Note that if we skip the *ignore_index* parameter while using the append function, we will get an error, as shown in the following:

CODE:

```
periodic_table.append({'Atomic Number':6,'Element':'Carbon','Symbol':'C'})
```

Output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-164-2e1fc586027a> in <module>
----> 1 periodic_table.append({'Atomic Number':6,'Element':'Carbon','Symbol':'C'})

~\Anaconda3\lib\site-packages\pandas\core\frame.py in append(self, other, ignore_index, verify_integrity, sort)
   6656                 other = Series(other)
   6657             if other.name is None and not ignore_index:
-> 6658                 raise TypeError('Can only append a Series if ignore_index=True'
   6659                                 ' or if the Series has a name')
   6660

TypeError: Can only append a Series if ignore_index=True or if the Series has a name
```

Using the *append* method, we can also add multiple rows by defining each row as a Series object and passing these Series objects as a list to the append method. The *pd.Series* method has a name attribute that assigns an index label to a Series.

CODE:

```
series1=pd.Series({'Atomic Number':7,'Element':'Carbon','Symbol':'C'},name=
len(periodic_table))
series2=pd.Series({'Atomic Number':8,'Element':'Oxygen','Symbol':'O'},name=
len(periodic_table)+1)
periodic_table.append([series1, series2])
```

Output:

	Atomic Number	Element	Symbol
0	1	Hydrogen	H
1	2	Helium	He
2	3	Lithium	Li
3	4	Beryllium	Be
4	5	Boron	B
5	7	Carbon	C
6	8	Oxygen	O

Note that we did not use the *ignore_index* parameter this time, since we have used the *name* parameter (refer to the error message shown earlier where it is mentioned that we can use either the *ignore_index* parameter or *name* parameter with the append method). Using the *name* parameter prevents the resetting of the index, which happens when we include the *ignore_index* parameter.

Understanding the various types of joins

Before we move on to the other methods for combining Pandas objects, we need to understand the concepts of an inner, outer, left, and right join. When you join two objects, the type of join determines which records from these objects get included in the final result set.

- Left join: All rows from the object on the left included in the combined object. Rows from the object on the right that match those from the left included.
- Right join: All rows from the object on the right included in the combined object. Rows from the object on the left that match those from the left included.
- Outer join: All rows from both objects included in the combined object (whether they match or not).
- Inner join: Only matching rows from both objects included.

Concat function (adding rows or columns from other objects)

This function gives us the option to add both rows and columns to a Pandas object. By default, it works on the row axis and adds rows.

Let us look at how the *concat* function works with an example. Here, we join two DataFrame objects vertically. The second DataFrame object is added after the last row of the first DataFrame object.

CODE:

```
periodic_table=pd.DataFrame({'Atomic Number':[1,2,3,4,5],'Element':
['Hydrogen','Helium','Lithium','Beryllium','Boron'],'Symbol':['H','He',
'Li','Be','B']})
```

```
periodic_table_extended=pd.DataFrame({'Atomic Number':[8,9,10],'Element':['
Oxygen','Fluorine','Neon'],'Symbol':['O','F','Ne']})
#Join these two DataFrames just created vertically using the concat
function:
pd.concat([periodic_table,periodic_table_extended])
```

Output:

	Atomic Number	Element	Symbol
0	1	Hydrogen	H
1	2	Helium	He
2	3	Lithium	Li
3	4	Beryllium	Be
4	5	Boron	B
0	8	Oxygen	O
1	9	Fluorine	F
2	10	Neon	Ne

We can also concatenate objects side-by-side along the column axis, as shown in the following.

CODE:

```
pd.concat([periodic_table,periodic_table_extended],axis=1)
```

Output:

	Atomic Number	Element	Symbol	Atomic Number	Element	Symbol
0	1	Hydrogen	H	8.0	Oxygen	O
1	2	Helium	He	9.0	Fluorine	F
2	3	Lithium	Li	10.0	Neon	Ne
3	4	Beryllium	Be	NaN	NaN	NaN
4	5	Boron	B	NaN	NaN	NaN

By default, the *concat* function performs an outer join, which returns all records of both the objects. The concatenated result set will have five records (equal to the length of the longer object – the first DataFrame). Since the second DataFrame has only three rows, you can see null values for the fourth and fifth row in the final concatenated object.

We can change this to an inner join by adding the *join* parameter. By using an inner join as shown in the following, the final result set with contain only those records from both the objects where the indices match.

CODE:

```
pd.concat([periodic_table,periodic_table_extended],axis=1,join='inner')
```

Output:

	Atomic Number	Element	Symbol	Atomic Number	Element	Symbol
0	1	Hydrogen	H	8	Oxygen	O
1	2	Helium	He	9	Fluorine	F
2	3	Lithium	Li	10	Neon	Ne

We can use the *keys* parameter to identify each of the objects that are being concatenated, in the final result set.

CODE:

```
pd.concat([periodic_table,periodic_table_extended],axis=1,keys=['1st
periodic table','2nd periodic table'])
```

Output:

	1st periodic table			2nd periodic table		
	Atomic Number	Element	Symbol	Atomic Number	Element	Symbol
0	1	Hydrogen	H	8.0	Oxygen	O
1	2	Helium	He	9.0	Fluorine	F
2	3	Lithium	Li	10.0	Neon	Ne
3	4	Beryllium	Be	NaN	NaN	NaN
4	5	Boron	B	NaN	NaN	NaN

Join method – index to index

The *join* method aligns two Pandas objects based on common index values. That is, it looks for matching index values in both objects and then align them vertically. The default type of join for this method is a left join.

Let us consider the following example where we join two objects.

CODE:

```
periodic_table.join(periodic_table_extended,lsuffix='_left',rsuffix='_right')
```

Output:

	Atomic Number_left	Element_left	Symbol_left	Atomic Number_right	Element_right	Symbol_right
0	1	Hydrogen	H	8.0	Oxygen	O
1	2	Helium	He	9.0	Fluorine	F
2	3	Lithium	Li	10.0	Neon	Ne
3	4	Beryllium	Be	NaN	NaN	NaN
4	5	Boron	B	NaN	NaN	NaN

Since the two DataFrame objects have common column names in the preceding example, we need to use the *lsuffix* and *rsuffix* parameters to differentiate between them. The indexes 0, 1, and 2 are common to both objects. The result set includes all the rows in the first DataFrame, and if there are rows with indices not matching in the second DataFrame, the value in all these rows is a null (denoted by NaN). The default join type used for the join method is a left join.

Merge method – SQL type join based on common columns

Like the *join* method, the *merge* method is also used to join objects horizontally. It is used when we join two DataFrame objects with a common column name. The main difference between the *join* and *merge* methods is that the *join* method combines the objects based on common index values, while the *merge* method combines the objects based on common column names. Another difference is that the default join type in the merge method is an inner join, while the join method performs a left join of the objects by default.

Let us look at how the merge method works with an example. The two DataFrame objects defined here have a column name that is common – Atomic Number. This is a scenario where we can apply the merge method.

CODE:

```
periodic_table=pd.DataFrame({'Atomic Number':[1,2,3,4,5],'Element':
['Hydrogen','Helium','Lithium','Beryllium','Boron'],'Symbol':['H','He',
'Li','Be','B']})
periodic_table_extended=pd.DataFrame({'Atomic Number':[1,2,3],'Natural':
'Yes'})
periodic_table.merge(periodic_table_extended)
```

Output:

	Atomic Number	Element	Symbol	Natural
0	1	Hydrogen	H	Yes
1	2	Helium	He	Yes
2	3	Lithium	Li	Yes

The presence of a common column name is essential for a merge operation, otherwise we would get an error. If there is a more than one column common between the two DataFrames, we can mention the column on which the merge is to be performed using the on parameter.

We can change the default join type (which is an inner join) using the *how* parameter.

CODE:

```
periodic_table.merge(periodic_table_extended,how='outer')
```

Output:

	Atomic Number	Element	Symbol	Natural
0	1	Hydrogen	H	Yes
1	2	Helium	He	Yes
2	3	Lithium	Li	Yes
3	4	Beryllium	Be	NaN
4	5	Boron	B	NaN

If we have the same column in both the objects that are being joined but their names are different, we can use parameters in the merge method to differentiate these columns.

In the following example, there are two columns with the same values but different names. In the first DataFrame object, the name of the column is 'Atomic Number', while in the second DataFrame object, the name of the column is 'Number'.

CODE:

```
periodic_table=pd.DataFrame({'Atomic Number':[1,2,3,4,5],'Element':
['Hydrogen','Helium','Lithium','Beryllium','Boron'],'Symbol':['H','He',
'Li','Be','B']})
periodic_table_extended=pd.DataFrame({'Number':[1,2,3],'Natural':'Yes'})
```

Using the *left_on* parameter for the column on the left, and *right_on* parameter for the column on the right, we merge the two objects as follows:

CODE:

```
periodic_table.merge(periodic_table_extended,left_on='Atomic Number',
right_on='Number')
```

Output:

	Atomic Number	Element	Symbol	Number	Natural
0	1	Hydrogen	H	1	Yes
1	2	Helium	He	2	Yes
2	3	Lithium	Li	3	Yes

Note that *append*, *merge*, and *join* are all DataFrame methods used with DataFrame objects while *concat* is a Pandas function.

Further reading:

Merge method: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html

Join method: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.join.html#pandas.DataFrame.join

Concat function: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html

Append method: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.append.html

Restructuring data and dealing with anomalies

As we have discussed earlier, data in its raw state is often messy and unfit for analysis. Most datasets require extensive treatment before they become fit for analysis and visualization. The most common problems encountered in datasets are given in the following.

- Data has missing values.
- Names of columns are not comprehensible.
- Variables are stored in rows as well as columns.
- A column may represent more than one variable.
- There are different observational units in a single table.
- There is data duplication.

- Data is structured around the wrong axis (for instance, horizontally instead of vertically).
- Variables are categorical, but we need them to be in a numeric format for performing calculations and visualizing them.
- The data type of a variable is not correctly recognized.
- Data contains spaces, commas, special symbols, and so on which need to be removed.

In the following sections, we understand how to handle missing and duplicate data, convert data from wide to long format, and how to use various methods like *pivot*, *stack*, and *melt*.

Dealing with missing data

Missing data in Pandas is represented by the value *NaN* (*Not a Number*), denoted as the keyword *np.nan*. We can use the *isna* or *isnull* method for finding null values. Both methods return a True (for a missing value) or False (for all other values) Boolean value for each object in the Series or DataFrame.

Let us see how many null values there are in the rainfall dataset.

CODE:

```
df=pd.read_csv('subset-covid-data.csv')
df.isna().sum().sum()
```

Output:

```
8
```

There are eight null values in this dataset. The *sum* method is used twice. The first sum method calculates the total number of missing values for each column, and the second sum method adds up these values to give the number of missing values in the entire DataFrame.

We have two options for dealing with this missing data - either we get rid of these values (drop them), or we substitute these values with a suitable measure (like the mean, median, or mode) that can be used as an approximation for the missing value. Let us look at each of these methods.

Dropping the missing data

The *dropna* method removes all the missing values in a DataFrame or Series.

CODE:

```
df.dropna()
```

Note that this method creates a copy of the data frame and does not modify the original DataFrame. To modify the original DataFrame, we use the inplace=True parameter.

CODE:

```
df.dropna(inplace=True)
```

Imputation

Imputation is the process of replacing missing values. In Pandas, there is an option to substitute the missing values with measures of central tendencies like the mean, median, or mode, using the *fillna* method. You can also fill the missing values with a fixed or constant value like 0.

We can use the forward fill technique to fill the missing value with the value just before it, or the backward fill technique to substitute the null value with the value just after it.

Using the same dataset (subset-covid-data.csv), let us try to understand the concepts of forward fill and backward fill.

CODE:

```
data=pd.read_csv('subset-covid-data.csv')
df=data[4:7]
df
```

As we can see, the DataFrame object, df (created from the original dataset), has missing values.

	country	continent	date	day	month	year	cases	deaths	country_code	population
4	Angola	Africa	2020-04-12	12	4	2020	0	0	AGO	30809762.0
5	Anguilla	America	2020-04-12	12	4	2020	0	0	NaN	NaN
6	Antigua_and_Barbuda	America	2020-04-12	12	4	2020	0	0	ATG	96286.0

Let us substitute the *NaN* value for the country `Anguilla' with the values preceding it (in the previous row), using the forward fill technique (*ffill*), as shown in the following.

CODE:

```
df.fillna(method='ffill')
```

Output:

	country	continent	date	day	month	year	cases	deaths	country_code	population
4	Angola	Africa	2020-04-12	12	4	2020	0	0	AGO	30809762.0
5	Anguilla	America	2020-04-12	12	4	2020	0	0	AGO	30809762.0
6	Antigua_and_Barbuda	America	2020-04-12	12	4	2020	0	0	ATG	96286.0

As we can see, the population field for Anguilla is substituted with the corresponding value of Angola (the record that precedes it).

We can also substitute the missing values using the backward fill technique, "bfill", which substitutes the missing values with the next valid observation that occurs in the row following it.

CODE:

```
df.fillna(method='bfill')
```

Output:

	country	continent	date	day	month	year	cases	deaths	country_code	population
4	Angola	Africa	2020-04-12	12	4	2020	0	0	AGO	30809762.0
5	Anguilla	America	2020-04-12	12	4	2020	0	0	ATG	96286.0
6	Antigua_and_Barbuda	America	2020-04-12	12	4	2020	0	0	ATG	96286.0

The missing population value for Anguilla is now substituted with the corresponding value from the next row (that of Antigua_and_Barbuda).

A standard method for imputation of missing values is to substitute the null values with the mean value of the other valid observations. The *fillna* method can be used for this purpose, as well.

Here, we are substituting the missing value in each column with the mean of the other two values in the column.

CODE:

```
df.fillna(df.mean())
```

Output:

	country	continent	date	day	month	year	cases	deaths	country_code	population
4	Angola	Africa	2020-04-12	12	4	2020	0	0	AGO	30809762.0
5	Anguilla	America	2020-04-12	12	4	2020	0	0	NaN	15453024.0
6	Antigua_and_Barbuda	America	2020-04-12	12	4	2020	0	0	ATG	96286.0

The missing population value for Anguilla is now substituted with the mean of the population figures for the other two countries (Angola &Antigua_and_Barbuda) in the DataFrame object, df.

Interpolation is another technique for estimating missing values in numeric columns, with the most straightforward interpolation technique being the linear interpolation method. In linear interpolation, the equation of a straight line is used to estimate unknown values from known values. If there are two points, (x_0, y_0) and $(x_1 y_1)$, then an unknown point (x,y) can be estimated using the following equation:

$$y - y_0 = \left(\frac{y_1 - y_0}{x_1 - x_0} \right) (x - x_0)$$

In Pandas, there is an *interpolate* method that estimates an unknown value from known values.

CODE:

```
df.interpolate(method='linear')
```

Output:

	country	continent	date	day	month	year	cases	deaths	country_code	population
4	Angola	Africa	2020-04-12	12	4	2020	0	0	AGO	30809762.0
5	Anguilla	America	2020-04-12	12	4	2020	0	0	NaN	15453024.0
6	Antigua_and_Barbuda	America	2020-04-12	12	4	2020	0	0	ATG	96286.0

The missing values in each column are interpolated using the other values in the column.

Further reading: See more on missing data in Pandas: https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html

Data duplication

Redundancy in data is a common occurrence with many records containing the same data.

Let us consider the following DataFrame:

CODE:

```
periodic_table=pd.DataFrame({'Atomic Number':[1,2,3,4,5,5],'Element':['Hydr
ogen','Helium','Lithium','Beryllium','Boron','Boron'],'Symbol':['H','He','L
i','Be','B','B']})
```

	Atomic Number	Element	Symbol
0	1	Hydrogen	H
1	2	Helium	He
2	3	Lithium	Li
3	4	Beryllium	Be
4	5	Boron	B
5	5	Boron	B

As we can see, there are duplicates in this data (the last two rows are the same).

In Pandas, the presence of duplicates can be detected using the *duplicated* method. The *duplicated* method returns a Boolean value for each row, as shown in the following. Since the fifth row is the duplicate of the fourth row, the Boolean value is True.

CODE:

```
periodic_table.duplicated()
```

Output:

```
0    False
1    False
2    False
3    False
```

```
4    False
5     True
dtype: bool
```

By chaining this method with the *sum* method, we can find the total number of duplicates in the DataFrame.

CODE:

```
periodic_table.duplicated().sum()
```

Output:

```
1
```

Let us now get rid of the duplicates using the *drop_duplicates* method.

CODE:

```
periodic_table.drop_duplicates(inplace=True)
```

Output:

	Atomic Number	Element	Symbol
0	1	Hydrogen	H
1	2	Helium	He
2	3	Lithium	Li
3	4	Beryllium	Be
4	5	Boron	B

The duplicate row has been removed. Since the *drop_duplicates* method does not make changes to the actual DataFrame object, we need to use the *inplace* parameter.

By default, the *drop_duplicates* method keeps the first row among all the duplicate rows. If we want to keep the last row, we can use the parameter *keep='last'*, as shown in the following.

CODE:

```
periodic_table.drop_duplicates(keep='last')
```

Output:

	Atomic Number	Element	Symbol
0	1	Hydrogen	H
1	2	Helium	He
2	3	Lithium	Li
3	4	Beryllium	Be
5	5	Boron	B

Apart from dealing with redundant or missing data, we may need to replace data that does not add value to our analysis, like blank spaces or special characters. Removal or replacement of values can be done using the replace method, discussed earlier. We may also need to change the data types of the columns since Pandas may not recognize all the data types correctly, which can be done using the "astype" method.

Tidy data and techniques for restructuring data

Tidy data is a term developed by Hadley Wickham. According to a paper authored by him (Link: `http://vita.had.co.nz/papers/tidy-data.pdf`), these are the three principles of tidy data:

1. Columns correspond to variables in the data, and each variable maps to a single column.
2. The rows contain only observations, not variables.
3. Each data structure or table contains only one observational unit.

Note that making data tidy is different from data cleansing. Data cleansing is concerned with dealing with missing values and redundant information, removing filler characters, and changing inaccurate data types. On the other hand, converting data to a tidy format involves restructuring the data and arranging it along the right axis, to facilitate ease of analysis.

Let us understand this with an example, using the following DataFrame.

	Attributes	Anita	Binni	Chaya	Dinara
0	Age	10	7	8	9
1	Height	122	135	142	129

The preceding DataFrame displays the age (in years) and height (in centimeters) of four students. Though this data is readable, it is not in the "tidy" form. There are three issues with this DataFrame that go against the principles of tidy data:

- The names of the students cannot be used as column names. Instead, we need to have a single variable for all the names.
- The attributes "Age" and "Height" should not be used as observations in rows. They are in fact separate variables and should be individual columns.
- There are two different observational units in the same DataFrame – years for measuring the age and centimeters for measuring the height

Data in the long format is considered to be tidy, and in the following section we will cover the methods in Pandas to convert a dataset to this structure.

Conversion from wide to long format (tidy data)

The following are two DataFrames, with the same data but having different structures (wide and long).

Wide Format

	Biology	Chemistry	Mathematics	Physics
Andrew	90	46	95	75
Sarah	87	56	74	65
Jason	45	87	45	33

Long Format

	student_name	subject	marks
0	Andrew	Biology	90
1	Andrew	Chemistry	46
2	Andrew	Mathematics	95
3	Andrew	Physics	75
4	Sarah	Biology	87
5	Sarah	Chemistry	56
6	Sarah	Mathematics	74
7	Sarah	Physics	65
8	Jason	Biology	45
9	Jason	Chemistry	87
10	Jason	Mathematics	45
11	Jason	Physics	33

The main benefit of converting to the long format is that this format facilitates ease of data manipulation, like adding or retrieving data, as we don't need to worry about the structure of the data. Also, data retrieval is significantly faster when data is stored in a long format.

Let us understand with this an example.

First, create the following DataFrame:

CODE:

```
grades=pd.DataFrame({'Biology':[90,87,45],'Chemistry':[46,56,87],'Mathematics':[95,74,45],'Physics':[75,65,33]},index=['Andrew','Sarah','Jason'])
grades
```

Output:

	Biology	Chemistry	Mathematics	Physics
Andrew	90	46	95	75
Sarah	87	56	74	65
Jason	45	87	45	33

In this DataFrame, we can see that the principles of tidy data are not followed. There are two primary variables (students and subjects) that have not been identified as columns. The values for the variable subjects, like Biology, Chemistry, Mathematics, and Physics, are observations and should not be used as columns.

Stack method (wide-to-long format conversion)

We can correct the anomalies observed in the grades DataFrame using the stack method, which takes all the column names and moves them to the index. This method returns a new DataFrame or Series, with a multilevel index.

CODE:

```
grades_stacked=grades.stack()
grades_stacked
```

Output:

```
Andrew  Biology        90
        Chemistry      46
        Mathematics    95
        Physics        75
Sarah   Biology        87
        Chemistry      56
        Mathematics    74
        Physics        65
Jason   Biology        45
        Chemistry      87
        Mathematics    45
        Physics        33
dtype: int64
```

As seen in the preceding output, the structure has been changed to a long format from a wide one.

Let us examine the data type of this stacked object.

CODE:

```
type(grades_stacked)
```

Output:

```
pandas.core.series.Series
```

As we can see, this is a Series object. We can convert this object to a DataFrame using the *reset_index* method so that the two variables – Name and Subject – can be identified as two separate columns:

CODE:

```
grades_stacked.reset_index()
```

Output:

	level_0	level_1	0
0	Andrew	Biology	90
1	Andrew	Chemistry	46
2	Andrew	Mathematics	95
3	Andrew	Physics	75
4	Sarah	Biology	87
5	Sarah	Chemistry	56
6	Sarah	Mathematics	74
7	Sarah	Physics	65
8	Jason	Biology	45
9	Jason	Chemistry	87
10	Jason	Mathematics	45
11	Jason	Physics	33

In the preceding output, we change the name of the columns using the *rename_axis* method and reset the index, as shown in the following.

CODE:

```
grades_stacked.rename_axis(['student_name','subject']).reset_
index(name='marks')
```

Output:

	student_name	subject	marks
0	Andrew	Biology	90
1	Andrew	Chemistry	46
2	Andrew	Mathematics	95
3	Andrew	Physics	75
4	Sarah	Biology	87
5	Sarah	Chemistry	56
6	Sarah	Mathematics	74
7	Sarah	Physics	65
8	Jason	Biology	45
9	Jason	Chemistry	87
10	Jason	Mathematics	45
11	Jason	Physics	33

To convert this DataFrame back to its original (wide) format, we use the unstack method, as shown in the following:

CODE:

```
grades_stacked.unstack()
```

Output:

	Biology	Chemistry	Mathematics	Physics
Andrew	90	46	95	75
Sarah	87	56	74	65
Jason	45	87	45	33

Melt method (wide-to-long format conversion)

In addition to the *stack* method, the *melt* method can also be used for converting data to the long format. The *melt* method gives more flexibility than the *stack* method by providing an option to add parameters for customizing the output.

Let us create the same DataFrame (in the wide format):

CODE:

```
grades=pd.DataFrame({'Student_Name':['Andrew','Sarah','Jason'],'Biology':
[90,87,45],'Chemistry':[46,56,87],'Mathematics':[95,74,45],'Physics':
[75,65,33]})
```

	Student_Name	Biology	Chemistry	Mathematics	Physics
0	Andrew	90	46	95	75
1	Sarah	87	56	74	65
2	Jason	45	87	45	33

Now, convert it into the long format using the *melt* method.

CODE:

```
grades.melt(id_vars='Student_Name',value_vars=['Biology','Chemistry',
'Physics','Mathematics'],var_name='Subject',value_name='Marks')
```

Output:

	Student_Name	Subject	Marks
0	Andrew	Biology	90
1	Sarah	Biology	87
2	Jason	Biology	45
3	Andrew	Chemistry	46
4	Sarah	Chemistry	56
5	Jason	Chemistry	87
6	Andrew	Physics	75
7	Sarah	Physics	65
8	Jason	Physics	33
9	Andrew	Mathematics	95
10	Sarah	Mathematics	74
11	Jason	Mathematics	45

We have used four variables in the melt method:

- id_vars: Column(s) that we don't want to reshape and preserve in its current form. If we look at the original wide format of the grades DataFrame, the Student_Name is correctly structured and can be left as it is.
- value_vars: Variable or the variables that we want to reshape into a single column. In the wide version of the grades DataFrame, there is a column for each of the four subjects. These are actually values of a single column.
- var_name: Name of the new reshaped column. We want to create a single column – "Subject", with the values "Biology", "Chemistry", "Physics", and "Mathematics".
- value_name: This is the name of the column ("Marks") containing the values corresponding to the values of the reshaped column ("Subject").

Pivot method (long-to-wide conversion)

The *pivot* method is another method for reshaping data, but unlike the *melt* and *stack* methods, this method converts the data into a wide format. In the following example, we reverse the effect of the melt operation with the pivot method.

CODE:

```
#original DataFrame
grades=pd.DataFrame({'Student_Name':['Andrew','Sarah','Jason'],'Biolo
gy':[90,87,45],'Chemistry':[46,56,87],'Mathematics':[95,74,45],'Physi
cs':[75,65,33]})
#Converting to long format with the wide method
grades_melted=grades.melt(id_vars='Student_Name',value_vars=['Biology','Che
mistry','Physics','Mathematics'],var_name='Subject',value_name='Marks')
#Converting back to wide format with the pivot method
grades_melted.pivot(index='Student_Name',columns='Subject',values='Marks')
```

Output:

Subject	Biology	Chemistry	Mathematics	Physics
Student_Name				
Andrew	90	46	95	75
Jason	45	87	45	33
Sarah	87	56	74	65

The following parameters are used with the pivot method:

- *index*: Refers to the column(s) to be used as an index. In the wide format of the grades DataFrame, we are using the Student_Name as the index.
- *columns*: Name of the column whose values are used to create a new set of columns. Each of the values of the "Subject" column forms a separate column in the wide format.

- *values*: The column used to populate the values in the DataFrame. In our grades example, the "Marks" column is used for this purpose.

Further reading:

Melt function: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html

Pivot function: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.pivot.html

Summary

1. Pandas, a Python library for data wrangling, offers a host of advantages, including support for a variety of input formats for data to be imported, integration with other libraries, speed, and functions for cleaning, grouping, merging, and visualizing data.
2. Series, DataFrames, and Indexes form the core of Pandas. A Series is one-dimensional and equivalent to a column, while a DataFrame is two-dimensional and equivalent to a table or spreadsheet. Series and DataFrames use indexes that are implemented using hash tables to speed up data retrieval.
3. There are various methods to create and modify Series and DataFrame objects. Python objects like lists, tuples, and dictionaries, as well as NumPy arrays, can be used to create Pandas objects. We can add and remove rows or columns, replace values, and rename columns.
4. Data retrieval in Pandas can be done by using either the position (*iloc* indexer) or the index label (*loc* indexer), or by specifying a condition (Boolean conditioning).
5. Pandas uses the *groupby* function for aggregation. Various functions can be used in conjunction with the *groupby* function to calculate aggregate measures for each group, filter the groups based on a condition, transform the values, or apply an operation on them.

6. Pandas provides support for combining, joining, and merging two or more objects. The *append* method adds an object to an existing object vertically, and the *concat* function can add objects either side by side or vertically. The *join* and *merge* methods join objects horizontally based on a common column or index values.

7. Tidy data refers to a structure where the columns correspond to variables in the dataset, the rows contain observations, and there is only one observational unit. Generally, data in the long (vertical) format is considered tidy. Functions like melt and stack convert a DataFrame into the long format. The unstack and pivot functions convert the data into the wide format.

Now that we have learned how to prepare data and make it ready for analysis, let us look at data visualization in the next chapter.

Review Exercises

Question 1
Write the function/indexer/attribute in Pandas for

- Importing data from an HTML file
- Exporting data to an Excel file
- Selecting data using the index position
- Selecting data using its label
- Replacing null values with the median
- Renaming columns
- Obtaining the number of rows and columns in a DataFrame
- Converting to the wide format using the pivot function
- Performing an inner join of two tables
- Changing the data type of a series

Question 2

The *df.describe* function returns the following summary statistical values: the count, minimum, maximum, standard deviation, percentiles, and means.

Which parameter would you add to obtain the following: the number of unique values, the value that occurs most frequently, and the frequency of this value?

Question 3

Import the "subset-covid-data.csv" in a DataFrame. Select the following data:

1. The country and continent columns
2. Set the 'country' column as the index and retrieve the population for the country "Algeria" using the at or loc indexers.
3. Select the value at the 50th row and the 3rd column using the iloc or iat indexers.
4. Retrieve the country code and population data for the last three records.
5. Select the data for the countries where the population is greater than 2.5 million, and the number of cases is greater than 3000.

Question 4

Import the data for the file "subset-covid-data.csv" in a DataFrame and write code statements for the following:

1. Deleting the "country_code" column
2. Inserting this column back at the same position that it was present earlier
3. Deleting the first three rows
4. Adding these rows back (at their original position)

Question 5

Create the following DataFrames:

`DataFrame name: orders_df`

	order_id	item
0	1	pens
1	2	shirts
2	3	coffee

`DataFrame name: orders1_df`

	order_id	item
0	4	crayons
1	5	tea
2	6	fruits

`DataFrame name:customers_df`

	order id	customer_name
0	1	anne
1	2	ben
2	3	carlos

Which function or method would you use to:

1. Combine the details of the first two DataFramesorders_df and orders1_df?
2. Create a DataFrame to show the customers and the items they ordered?

3. Make the order_id column as the index for orders_df and customers_df? Which method would you now use to combine these two objects to show which orders were placed by customers?

Question 6

The following DataFrame records the weight fluctuations of four people:

	Anna	Ben	Carole	Dave
0	51.0	70.0	64.0	81.0
1	52.0	70.5	64.2	81.3
2	51.4	69.1	66.8	80.5
3	52.8	69.8	66.0	80.9
4	50.5	70.5	63.4	81.4

1. Create the preceding DataFrame.
2. Convert this DataFrame into a tidy format.
3. Determine who among these four people had the least fluctuation in weight.
4. For people whose average weight is less than 65 kgs, convert their weight (on all four days) into pounds and display this data.

Question 7

The object data type is used for columns that contain the following data:

1. Strings
2. Numbers (int and float)
3. Booleans
4. A combination of all the preceding data types

Question 8

A column can be accessed using the dot notation (a.column_name) as well as the indexing operator (a[column_name]). Which is the preferred notation, and why?

Question 9

Which method is used to obtain the metadata in a DataFrame?

1. Describe method
2. Value_counts method
3. Info method
4. None of the above

Question 10 (fill in the blanks)

- The default join type for the join method is ____ and the parameter for adding a join type is ____
- The default join type for the merge method is ____ and the parameter for adding a join type is ____
- The default join type for the concat function is ____ and the parameter for adding a join type is ____
- The function in Pandas that created an object representing a date/time value and is equivalent to the following Python functions: datetime.date, datetime.time, or datetime.datetime is ____
- The function in Pandas equivalent to the datetime.timedelta function in Python for representing a duration of time is ____

Answers

Question 1

- Importing data from an HTML file:

  CODE:

  ```
  pd.read_html(url)
  ```

- Exporting data to an Excel file:

  CODE:

  ```
  df.to_excel(name_of_file)
  ```

- Selecting data using the index position:

  CODE:

  ```
  df.iloc[index position]
  ```

- Selecting data using its label:

  CODE:

  ```
  df.loc[column name or index label]
  ```

- Replacing null values in a DataFrame with the median:

  CODE:

  ```
  df.fillna(df.median())
  ```

- Renaming columns:

  ```
  df.rename(columns={'previous_name:'new_name'})
  ```

- Obtaining the number of rows and columns in a DataFrame:

  CODE:

  ```
  df.shape
  ```

- Converting to the wide format using the *pivot* function:

  CODE:

  ```
  df.pivot(index=column_name1,
  columns=column_name2, values=column_name3)
  ```

- Performing an inner join of two DataFrame objects

  CODE:

  ```
  df1.merge(df2)
  ```

  The default join type is *inner* for the *merge* method; hence we need not explicitly mention it.

- Changing the data type of a Series object/column:

  CODE:

  ```
  series_name.astype(new data type)
  ```

Question 2

We can obtain all three values using the include='all' parameter, as shown in the following:

CODE:

```
df.describe(include='all')
```

Question 3

CODE:

```
df=pd.read_csv('subset-covid-data.csv')
```

1.Retrieving the country and continent columns

CODE:

```
df[['country','continent']]
```

2.Set the 'country' column as the index and retrieve the population for the country "Algeria" using the at or loc indexers.

CODE:

```
df.set_index('country',inplace=True)
```

Retrieve the population using this statement:

CODE:

```
df.at['Algeria','population']
```

or

```
df.loc['Algeria','population']
```

3.Select the value at the 50th row and the 3rd column using the iloc or iat indexers.

```
df.iat[49,2]
```

Or

```
df.iloc[49,2]
```

4.Retrieve the country code and population data for the last three records:

CODE:

```
df.iloc[203:,-1:-3:-1]
```

Or

CODE:

```
df.iloc[203:,7:]
```

5.Select the data for the countries where the population is greater than 2.5 million, and the number of cases is greater than 3000.

CODE:

```
df[(df['cases']>=3000) & (df['population']>=25000000)]
```

Question 4 (addition and deletion of rows and columns)

1. Delete the "country_code" column:

   CODE:

   ```
   x=df.pop('country_code')
   ```

2. Insert this column back at the same position that it was present earlier:

   CODE:

   ```
   df.insert(8,'country_code',x)
   ```

3. Delete the first three rows:

   CODE:

   ```
   df.drop([0,1,2],inplace=True)
   ```

4. Add these rows back to the DataFrame at their original positions:

   CODE:

   ```
   x=df.iloc[0:3]
   pd.concat([x,df])
   ```

Question 5

Create the following DataFrames:

CODE:

```
orders_df=pd.DataFrame({'order_id':[1,2,3],'item':['pens','shirts',
'coffee']})
orders1_df=pd.DataFrame({'order_id':[4,5,6],'item':['crayons','tea',
'fruits']})
customers_df=pd.DataFrame({'order id':[1,2,3],'customer_name':['anne',
'ben','carlos']})
```

Functions for combining objects

- Combine the details of the two DataFrames orders_df and orders1_df:

  CODE:

  ```
  pd.concat((orders_df,orders1_df))
  ```

- Create a combined DataFrame to show the customers and the items they ordered:

  CODE:

  ```
  pd.merge(orders_df,customers_df,left_on='order_id',right
  _on='order id')
  ```

- Make the order_id/order id column as the index for the "orders_df" and "customers_df" DataFrames:

  CODE:

  ```
  orders_df.set_index('order_id',inplace=True)
  customers_df.set_index('order id',inplace=True)
  ```

- Method to combine these two objects to show which orders were placed by customers:

  CODE:

  ```
  orders_df.join(customers_df)
  ```

Question 6 (tidy data and aggregation)

1. Create this DataFrame.

   CODE:

   ```
   df=pd.DataFrame({'Anna':[51,52,51.4,52.8,50.5],'Ben':
   [70,70.5,69.1,69.8,70.5],'Carole':[64,64.2,66.8,66,63.4],
   'Dave':[81,81.3,80.5,80.9,81.4]})
   ```

2. Convert this into a tidy format.

   We use the *melt* method to convert the DataFrame to a long format and then rename the columns.

   CODE:

   ```
   df_melted=df.melt()
   df_melted.columns=['name','weight']
   ```

3. In order to find out who among these four people had the least fluctuation in weight with respect to their mean weight, we need to first aggregate the data.

   We use the *groupby* function to group the data according to the name of the person and use the standard deviation (*np.std*) aggregation function.

   CODE:

   ```
   df_melted.groupby('name').agg({'weight':np.std})
   ```

   Output:

	weight
name	
Anna	0.893308
Ben	0.580517
Carole	1.446375
Dave	0.356371

   Dave had the least standard deviation as seen from the preceding output, and therefore, the least fluctuation in his weight.

4. For people whose average weight is less than 65 kgs, we now need to convert their weight to pounds and display their weight for all the four days.

   We use the filter method to filter out the groups where the average weight is greater than 65 kgs using the filter method and then apply the transform method to convert the weight into pounds.

   CODE:

   ```
   df_melted.groupby('name').filter(lambda x:x.mean()>65)
   ['weight'].transform(lambda x:float(x)*2.204)
   ```

Question 7

Options 1 and 4

The object data type in Pandas is used for strings or columns that contain mixed data like numbers, strings, and so on.

Question 8

The indexing operator [] is preferred since it does not clash with built-in methods and works with column names that have spaces and special characters. The dot notation does not work when you have to retrieve a column that contains spaces.

Question 9

Option 3 – info method.

The info method is used to obtain the metadata for a Series or DataFrame object. It gives us the name of the columns, their data types, and number of non-null values and the memory usage.

Question 10 (fill in the blanks)

The default join type for the join method is a "*left join*" and the parameter for adding a join type is the "*how*" parameter

The default join type for the merge method is an *"inner join"* and the parameter for adding a join type is the *"how"* parameter

The default join type for the concat function is an *"outer join"* and the parameter for adding a join type is the *"join"* parameter

The function in Pandas that created an object representing a date/time value and is equivalent to the following Python functions: *datetime.date*, *datetime.time*, or *datetime.datetime* is *pd.Timestamp*

The function in Pandas equivalent to the *datetime.timedelta* function in Python for representing a duration of time is *pd.Timedelta*